O chatGPT forneceu o seguinte código:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void gaussElimination(double *matrix, double *b, int n, int rank, int
size) {
    for (int k = 0; k < n; ++k) {
        if (rank == k % size) {
            // Normalizar linha pivô
            double pivot = matrix[k * n + k];
            for (int j = k; j < n; ++j) {
                matrix[k * n + j] /= pivot;
            }
            b[k] /= pivot;
        }

        // Broadcast da linha pivô
        MPI_Bcast(&matrix[k * n], n, MPI_DOUBLE, k % size,
MPI_COMM_WORLD);
        MPI_Bcast(&b[k], 1, MPI_DOUBLE, k % size, MPI_COMM_WORLD);

        // Eliminação das linhas subsequentes
        for (int i = k + 1; i < n; ++i) {
            if (i % size == rank) {
                double factor = matrix[i * n + k];
                for (int j = k; j < n; ++j) {
                    matrix[i * n + j] -= factor * matrix[k * n + j];
                }
                b[i] -= factor * b[k];
            }
        }
    }
}

void backSubstitution(double *matrix, double *b, double *x, int n, int
rank, int size) {
    for (int i = n - 1; i >= 0; --i) {
        if (rank == i % size) {
            x[i] = b[i];
            for (int j = i + 1; j < n; ++j) {
                x[i] -= matrix[i * n + j] * x[j];
            }
        }
        MPI_Bcast(&x[i], 1, MPI_DOUBLE, i % size, MPI_COMM_WORLD);
    }
}

int main(int argc, char *argv[]) {
    int n = 3; // Dimensão do sistema (modifique conforme necessário)
```

```
    double matrix[] = {
        2.0, 1.0, -1.0,
        -3.0, -1.0, 2.0,
        -2.0, 1.0, 2.0
    };
    double b[] = {8.0, -11.0, -3.0};
    double x[3];

    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    gaussElimination(matrix, b, n, rank, size);
    backSubstitution(matrix, b, x, n, rank, size);

    if (rank == 0) {
        printf("Solução do sistema:\n");
        for (int i = 0; i < n; ++i) {
            printf("x[%d] = %f\n", i, x[i]);
        }
    }

    MPI_Finalize();
    return 0;
}
```

A primeira coisa que fizemos foi adicionar a função MPI_Wtime() para marcar o tempo de execução.

```
tempo_inicial = MPI_Wtime();
gaussElimination(matrix, b, n, rank, size);
backSubstitution(matrix, b, x, n, rank, size);
tempo_final = MPI_Wtime();
```

Os tempos iniciais foram muito pequenos, porém os tempos para mais processos estava dando maior. Isso se dá pelo overhead de comunicação, a própria divisão de processos acabou usando mais tempo:

```
$ mpirun -np 2 ./gauss_mpi_melhorado
Solução do sistema:
x[0] = 2.000000
x[1] = 3.000000
x[2] = -1.000000
Foram gastos 0.0000430000 segundos

$ mpirun -np 4 ./gauss_mpi_melhorado
Solução do sistema:
x[0] = 2.000000
```

```
x[1] = 3.000000
x[2] = -1.000000
Foram gastos 0.0000680000 segundos

$ mpirun -np 8 ./gauss_mpi_melhorado
Solução do sistema:
x[0] = 2.000000
x[1] = 3.000000
x[2] = -1.000000
Foram gastos 0.0001970000 segundos
```

Então, ao invés de criar uma pequena matriz estática, criamos dinamicamente uma matriz de 1000 valores. Para isso, usamos malloc para armazenar essa matriz.

```
    int n = 2000; // Tamanho do sistema
    double *matrix, *b, *x;

    // Alocação dinâmica dos endereços para a matriz
    matrix = (double *)malloc(n * n * sizeof(double));
    b = (double *)malloc(n * sizeof(double));
    x = (double *)malloc(n * sizeof(double));

    if (!matrix || !b || !x) {
        printf("Erro na alocação de memória\n");
        exit(1);
    }

    // Preenche matriz e vetor com valores aleatórios
    srand(time(NULL));
    for (int i = 0; i < n; ++i) {
        b[i] = rand() % 100; // Valores no vetor b
        for (int j = 0; j < n; ++j) {
            matrix[i * n + j] = (i == j) ? rand() % 100 + 1 : rand() %
100;
            // Matriz diagonal dominante
        }
    }
```

Para conferir se o resultado encontrado para a matriz x está correto, comparamos ax (matriz inicial * resultado encontrado) com o b aleatório inicial:

```
// Verificando a precisão da solução
double error = 0.0;
for (int i = 0; i < n; ++i) {
    double ax = 0.0;
    for (int j = 0; j < n; ++j) {
        ax += matrix[i * n + j] * x[j]; // calcula os resultados para b
com x encontrado
```

```
    }
    error += fabs(ax - b[i]); // compara o valor com o b aleatório
inicial
}
printf("Erro total: %e\n", error);
```